

**AFRL-IF-RS-TR-2001-281**  
**Final Technical Report**  
**January 2002**



# **EVENTS AND RULES FOR JAVA: DESIGN AND IMPLEMENTATION OF A SEAMLESS APPROACH**

**University of Florida**

*APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.*

**20020507 105**

**AIR FORCE RESEARCH LABORATORY  
INFORMATION DIRECTORATE  
ROME RESEARCH SITE  
ROME, NEW YORK**

This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

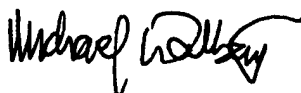
AFRL-IF-RS-TR-2001-281 has been reviewed and is approved for publication.

APPROVED:



RAYMOND A. LIUZZI  
Project Engineer

FOR THE DIRECTOR:



MICHAEL TALBERT, Maj., USAF, Technical Advisor  
Information Technology Division  
Information Directorate

If your address has changed or if you wish to be removed from the Air Force Research Laboratory Rome Research Site mailing list, or if the addressee is no longer employed by your organization, please notify AFRL/IFTD, 525 Brooks Road, Rome, NY 13441-4505. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188	
<small>Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.</small>				
1. AGENCY USE ONLY (Leave blank)		2. REPORT DATE JANUARY 2002		3. REPORT TYPE AND DATES COVERED Final Sep 98 - Sep 01
4. TITLE AND SUBTITLE EVENTS AND RULES FOR JAVA: DESIGN AND IMPLEMENTATION OF A SEAMLESS APPROACH			5. FUNDING NUMBERS C - F30602-98-1-0291 PE - 62234N PR - R427 TA - 00 WU - PX	
6. AUTHOR(S) S. Chakravarthy and R. Dasari				
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) University of Florida PO Box 116120 Gainesville Florida 32611-6120			8. PERFORMING ORGANIZATION REPORT NUMBER  N/A	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Air Force Research Laboratory    Space & Naval Wafare Systems Center 525 Brooks Road                      53560 Hull Street Rome New York 13441-4505        San Diego California 92152-5001			10. SPONSORING/MONITORING AGENCY REPORT NUMBER  AFRL-IF-RS-TR-2001-281	
11. SUPPLEMENTARY NOTES Air Force Research Laboratory Project Engineer: Raymond A. Liuzzi/IFTD/(315) 330-3577				
12a. DISTRIBUTION AVAILABILITY STATEMENT APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.			12b. DISTRIBUTION CODE	
13. ABSTRACT (Maximum 200 words) This report investigates the re-designing and implementation of an active database subsystem in the JAVA environment. This was accomplished to overcome some of the limitations of the C++ environment in providing active capabilities to passive systems. Next to exploit some of the capabilities provided by the JAVA environment that would be applicable and used for an active system. Finally this provides a means for an active capability in the JAVA environment, as more and more OODBMS (Poet, ObjectStore etc.) and Distributed Systems are being developed in JAVA.				
14. SUBJECT TERMS Computers, Software, Data Bases, Knowledge Bases, Artificial Intelligence			15. NUMBER OF PAGES 48	
			16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT  UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE  UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT  UNCLASSIFIED	20. LIMITATION OF ABSTRACT  UL	

# Contents

List of Figures .....	iii
EVENTS AND RULES FOR JAVA: DESIGN AND IMPLEMENTATION OF A SEAMLESS APPROACH.....	iv
Abstract .....	iv
1. INTRODUCTION .....	1
2. ECA RULE SEMANTICS .....	3
2.1. Primitive Events .....	4
2.2. Composite Events .....	4
2.3. Parameter Contexts .....	5
2.4. Coupling Modes .....	6
2.5. Trigger Modes .....	7
2.6. Rule Priority .....	8
3. OVERVIEW OF RELATED WORK.....	8
3.1. Ode .....	8
3.2. ADAM .....	9
3.3. Samos .....	10
3.4. Vitria BusinessWare Process Automator .....	11
3.5. WebLogic Events .....	11
3.6. Sentinel .....	12
4. DESIGN OF THE EVENT DETECTOR.....	13
4.1. Types of Events .....	13
4.2. Event Parameters .....	15
4.3. Rules .....	15
4.4. Instance Level Rules .....	17
4.5. Class Hierarchy .....	17
4.6. Event Graph .....	18
4.6.1 Primitive Event Node.....	19
4.6.2 Composite Event Node .....	20
4.6.3 Event Table .....	20
4.6.4 Rule Node .....	21
4.7. C++ Versus Java Approach .....	21
4.7.1 Representation of a Rule .....	21
4.7.2 Temporal Events .....	21
4.7.3 Vector and Hash table Data Types in Java .....	22
5. IMPLEMENTATION.....	22
5.1. Primitive Event Detection.....	23
5.2. Composite Event Detection .....	26
5.3. Parameter Lists.....	27

5.4.	Detecting Temporal Events.....	28
5.5.	Temporal Event Handler.....	29
5.6.	Event Detector as a Thread .....	31
5.7.	Rule Scheduling.....	33
6.	CONCLUSIONS AND FUTURE WORK.....	35
6.1.	Future Work.....	36
	References.....	37

## List of Figures

Figure 1 Class Hierarchy .....	18
Figure 2 An Example Event Graph.....	19
Figure 3 An Instance list associated with a primitive event .....	20
Figure 4 An Event Table.....	21
Figure 5 The Event Detector Thread .....	24
Figure 6 An Event Detector Graph.....	25
Figure 7 The data structure used for storing the parameter .....	27
Figure 8 Processing Event Notifications from Event Detector Thread .....	31
Figure 9 Rule-list data structure.....	33

# EVENTS AND RULES FOR JAVA: DESIGN AND IMPLEMENTATION OF A SEAMLESS APPROACH

## Abstract

During the last decade, the paradigm of object-oriented software development has become one of the central means of developing software systems. In the object-oriented paradigm, applications are modeled as a set of interacting objects that require and provide services. This approach assumes that each object knows where the service is and calls it directly. However, this call-driven approach is not the only manner to invoke behavior. The proponents of active systems have proposed event-condition-action (ECA) rules, a mechanism where behavior is invoked automatically as a response to events but without user or application intervention. An active system automatically monitors events, evaluates conditions defined over the state of the system when the events occur, and invokes the action associated with the event-condition pair based on the result of condition evaluation. Making a passive system active requires an expressive event specification language with well-defined semantics, algorithms for detecting the events, designing an event detector and implementing it. The environment (the programming language and the operating system) in which a system is built influences how the event detector is designed and implemented.

Sentinel, developed at University of Florida, provides active capability to Open OODB, an object-oriented database management system (OODBMS) that was implemented in C++. However, C++ environment had certain limitations that proved deterrent to implementing some of the features of active capability. We tried to overcome these limitations by redesigning the active subsystem in Java, the most popular language now. Although both C++ and Java are both object-oriented languages, they differ in some of the capabilities they offer such as support for system calls, obtaining information of the application objects during run-time etc.

This report discusses the re-designing and implementation of the active subsystem in the Java environment. There are three motivations behind our objective of re-designing and implementing the active subsystem in the Java environment. First, we would like to overcome some of the limitations of the C++ environment in providing active capability to passive systems. Second, we would like to exploit some of the capabilities provided by the Java environment that would be very much applicable and useful for an active system. Finally, we feel there is a need for active capability in the Java environment, as more and more OODBMSs (Poet, ObjectStore etc.) and Distributed Systems (OrbixWeb) are being developed in Java.

## 1. INTRODUCTION

During the last decade, many software systems have been built using the object-oriented paradigm. In these systems, applications are modeled as a set of interacting objects that require and provide services. These systems are mostly passive since any situation to be monitored in the system has to be done explicitly by the user or the application by polling the system at certain intervals. For example, in a hospital environment if the electrocardiogram readings are recorded in the system for an intensive care unit patient, it is the responsibility of the doctor or nurse to check for the change of values over a period of time to determine any state of emergency. On the other hand, an active system can continuously monitor situations to initiate appropriate actions in response to updates, occurrence of particular states or transition of states automatically, possibly subject to timing constraints. An active system consists of event-condition-action or ECA rules that are used to invoke actions as a response to events occurring in the application without user or application intervention.

Situation monitoring can be done by defining ECA rules on those events that are of interest in the system. An event is defined as an instantaneous and atomic (happens completely or not at all) point of occurrence within an application. An ECA rule consists of three components – an event, a condition, and an action. According to the ECA rule semantics [1], whenever an event occurs, a condition is checked and the action is carried out if the condition evaluates to true. The behavior exhibited by applications by means of ECA rules, i.e., an action being carried out as a consequence of a certain event is known as active behavior. Programming languages, database systems and graphical user interfaces are being enhanced to provide explicit support for active behavior due to the large range of applications that naturally express their semantics using this paradigm. Active behavior is also useful to those applications that require monitoring situations and reacting to them without user or application intervention. Initially, active capability was incorporated into relational DBMSs by means of triggers. With the advent of OODBMSs, there was a need for extending the active capability from the relational domain to the object-oriented domain to support both OODBMSs and object-oriented applications.

Broadly, events in a system can be classified into two types – *internal events* and *external events*. Internal events are detected by the system/application whereas external events occur and are detected outside the system but are relevant to the system/application under consideration. Typical external events include the user, the operating system, or the user-interface part of the application (when the system of interest is the non-user-interface part of the application). The user can be interested in undertaking an action linked to a certain event or a set of events, such as showing a new window when the mouse is moved over a certain rectangle of the display, or updating the valid domain of  $x$  when  $y$  is changed and both are related in some way.

The process of incorporating active behavior entails the following steps: (1) defining the event and the rules associated with the event; (2) detecting the event when it occurs; (3) reacting to the event, i.e., carrying out the operations specified in the actions (rules) defined over the event. It should be noted that an event is associated with a set of parameters that are passed to the condition and action parts of the rule. The condition and



action parts can utilize these parameters in their execution. [2] Classifies systems based upon how and where event detection and action executions are done. According to this classification, software systems can be *Active*, *Event-driven*, *Event source* or *Call-driven*.

- **Active Systems** – These systems support both event detection and action execution (i.e., react *automatically* to events). The response is achieved through the run-time support part of the development system, according to the rule definitions given by the application. Examples are active database management systems; constraint based programming systems, etc.
- **Event-driven Systems** – These systems support only event detection. The application can explicitly request about events detected by the system, but the application is responsible for consuming the event and carrying out the action. This is usually achieved by continuously polling the system or by callback mechanisms. In a callback mechanism, the application is awakened at a certain point of execution when the interesting event happens, much like interruptions used to signal external events. Examples include most Graphical User Interface (GUI) subsystems.
- **Event source Systems** – In event source systems, the application detects the event and signals it to the run-time part of the development system or to another application. The later system perceives the event as an external event. Examples include security alerter in computer-controlled machines and hardware drivers.
- **Call-driven Systems** – These systems do not support event detection. The application has to detect relevant circumstances and execute the actions.

The process of augmenting an existing passive system to support active capability involves the specification of events and rules, rule management and rule execution. The environment into which ECA rules are incorporated has a bearing on some of the above. As described in Anwar et al. [3], event detection is considerably complex for an object-oriented environment and furthermore, compile time and runtime issues need to be addressed. Sentinel, developed at the University of Florida [4], provides active capability to OpenOODB, an object-oriented DBMS that does not have active capability. Sentinel supports an expressive event specification language (termed Snoop [5]), a Local Event Detector that detects events and a Rule Scheduler that schedules the rules and executes them. In the process of providing active capability to an OODBMS, Sentinel also addresses the fundamental differences between providing active capability to relational database systems and object-oriented database systems [6]. Although OODBMSs are better in providing support for complex data types when compared to relational DBMSs, providing active capability to OODBMSs is more complex than providing active capability in relational DBMSs.

Owing to the C++ implementation of OpenOODB, all the components of Sentinel were developed in C++. As already mentioned, the C++ environment had a bearing on the active behavior supported, like the types of event parameters allowed, how the condition and action parts of the rule were modeled etc. Since C++ does not have a mechanism to obtain the type information of a variable at run-time, only known data types such as primitive types could be passed as event parameters. In C++, condition and action parts of the rule were modeled as C functions rather than C++ methods since it was not possible to have a reference to a method of a class. The reference is required in

order to execute the condition and action parts of the rule when the event is detected. But in an object-oriented environment, it would be more useful to execute conditions and actions on a class or an object rather than as stand-alone functions. That way, conditions and actions can access the attributes defined in the class. These were some of the limitations of providing active capability in a C++ environment. This report addresses the issues involved in the redesign of the subsystem that provides active capability while moving from the C++ environment to the Java environment, as well as overcoming the limitations of the C++ environment. Although both C++ and Java are object-oriented programming languages, there are some differences in ideologies and capabilities provided by them. For example, it is possible to call some of the services of the underlying operating system directly from a C++ application whereas Java applications cannot directly make any calls to the operating system. As another example, Java provides mechanisms to obtain some dynamic information about the application objects during runtime whereas C++ does not provide any such mechanism.

There are several motivations behind our objective of re-designing and implementing the active subsystem in the Java environment. First, we would like to overcome the limitations of C++ environment in providing active capability. Second, we would like to exploit some of the dynamic capabilities provided by the Java environment that would be very much applicable and useful for an active system. Finally, we feel there is a need for active capability in the Java environment, as more and more OODBMSs (Poet, ObjectStore etc.) and Distributed Systems (OrbixWeb) are being developed in Java.

The aim of this report is to design and develop a system that provides support for events and rules in Java applications (whether it is a DBMS developed in Java or a general Java application) in a *seamless* manner. This report discusses the architecture, design, and implementation of such a system.

## 2. ECA RULE SEMANTICS

The design of an active system involves a method for specifying events and composite event expressions with associated semantics. A primitive event is an event expression comprising a single event in the domain of consideration, and a composite event is an event expression comprising a set of constituent primitive events connected through one or more composite event operators. Furthermore, composite events can be detected in different parameter contexts. This section summarizes the various composite event operators, the semantics associated with them and the different *parameter contexts* a composite event is detected, as described in Snoop [7, 8]. Rules can be defined on both primitive and composite events. Whenever an event (primitive or composite) is detected, the rules associated with that event are executed. Also, rules can be specified with a priority, a coupling mode and a triggering mode. This section also discusses the different coupling modes and triggering modes associated with a rule as well as rule priority.

## 2.1. Primitive Events

According to Krishnaprasad [9], events are specific points of interest on the time line and an event expression defines an interval on the time line. An event is an instantaneous and atomic (happening completely or not at all) occurrence. What can be defined as an event can be different in different domains. For example, in the relational database domain, database operations such as retrieve, insert, update, and delete can be defined as events. In the object-oriented domain, method invocations can be defined as events. An event is said to occur whenever a method is invoked. Since the execution of a method spans an interval of time whereas an event is an instantaneous occurrence, it is necessary to further identify a point of interest within the method. Event modifiers were introduced to transform an interval (typically formed by a method or procedure execution) to one or more events, each of which corresponds to a point of interest within the closed interval defined by the method. The *begin* event modifier denotes an event occurrence at the beginning of the method and the *end* event modifier denotes an event occurrence at the end of the method.

## 2.2. Composite Events

Two or more primitive events can be combined to form a composite event expression by applying one or more of the event operators defined in snoop. In the absence of event operators, several rules may be required to specify a composite event. In most cases, it may not be possible to model a composite event in this manner. Sentinel defines a composite event as an event obtained by the application of an event modifier to a composite event expression. By default, the *end* event modifier is assumed.

A composite event expression is defined recursively, as an event expression formed by using a set of primitive event expressions, event operators, and composite event expressions constructed up to that point. A composite event that is defined using the snoop operators NOT,  $\wedge$  (AND) and + (PLUS) looks as follows:

$$\text{NOT } (E_1, (E_2 \wedge E_3), (E_1 + "5 \text{ sec}"))$$

As indicated in the above expression, it is possible to nest composite event expressions. In addition to the nesting of composite events, it is also possible to use an event sub-expression in more than one event expression. Below, each composite event operator and its semantics is briefly described. The upper case letter E is used to represent an event type and the lower case letter e is used to represent an instance of the event E. Superscripts are used to denote the relative time of occurrence with respect to events of the same type. An event E (primitive or composite) is a function from the time domain onto the Boolean values, True and False. The function is given by

$$E(t) = \text{True if an event of type E occurs at time point } t$$

*False otherwise*

The negation of an event is given by  $\sim E$ . Given a time point, it computes the non-occurrence of an event at that point. The Snoop event operators and the semantics of composite events formed by these event operators are as follows-

- **OR ( $\vee$ ):** Disjunction of two events  $E_1$  and  $E_2$  denoted by  $E_1 \vee E_2$ , occurs when either  $E_1$  occurs or  $E_2$  occurs.
- **AND ( $\wedge$ ):** Conjunction of two events  $E_1$  and  $E_2$ , denoted by  $E_1 \wedge E_2$  occurs when both  $E_1$  and  $E_2$  occur, irrespective of their order of occurrence.
- **SEQUENCE (;):** Sequence of two events  $E_1$  and  $E_2$ , denoted by  $E_1;E_2$  occurs when  $E_2$  occurs provided  $E_1$  has already occurred. This implies that the time of occurrence of  $E_1$  is guaranteed to be less than the time of occurrence of  $E_2$ .
- **NOT ( $\neg$ ):** The *not* operator, denoted by  $\neg(E_2)[E_1,E_3]$  detects the non-occurrence of the event  $E_2$  in the closed interval formed by  $E_1$  and  $E_3$ .
- **Aperiodic Event Operators ( $A, A^*$ ):** The Aperiodic operator  $A$  allows one to express the occurrence of an aperiodic event in the half-open interval formed by  $E_1$  and  $E_3$ . There are two variations of this event specification. The non-cumulative variant of an aperiodic event is expressed as  $A(E_1, E_2, E_3)$ , where  $E_1, E_2$  and  $E_3$  are arbitrary events. The event  $A$  is signaled each time  $E_2$  occurs during the half-open interval defined by  $E_1$  and  $E_3$ .  $A$  can occur zero or more times (zero times either when  $E_2$  does not occur in the interval or when no interval exists for the definitions of  $E_1$  and  $E_3$ ).

There are situations when a given event is signaled more than once during a given interval (e.g. within a transaction), but rather than detecting the event and firing the rule every time the event occurs, the rule has to be fired only once. To meet this requirement, there is an operator  $A^*(E_1, E_2, E_3)$  that occurs only once when  $E_3$  occurs and accumulates the occurrences of  $E_2$  in the half-open interval formed by  $E_1$  and  $E_3$ . This constructor is useful for integrity checking in databases and for collecting parameters of an event over an interval for computing aggregates. As an example, the highest or lowest stock price can be computed over an interval using this operator.

In this formulation  $E_2$  is not included because the occurrence of the composite event  $A^*$  which coincides with the occurrence of  $E_3$  is not constrained by the occurrence of  $E_2$ . However, the parameters of  $A$  will contain the parameters of  $E_2$ .

- **Periodic Event Operators ( $P, P^*$ ):** A periodic event is an event  $E$  that repeats itself within a constant and finite amount of time. Only a time specification is meaningful for  $E$ . The notation used for expressing a periodic event is  $P(E_1, [t], E_3)$  where  $E_1$  and  $E_3$  are events and  $t$  is the time specification.  $P$  occurs for every  $t$  in the half-open interval  $(E_1, E_3]$ .  $t$  is assumed to be positive.

### 2.3. Parameter Contexts

As mentioned before, composite events can be detected in more than one parameter context. The notion of parameter contexts was primarily introduced for the purpose of capturing application semantics while computing the parameters of composite events when they are not unique. They serve the purpose of disambiguating the parameter computation and at the same time accommodate a wide range of application requirements [9]. It is to be noted that the semantics of a primitive event is identical in all contexts. The parameter contexts proposed by Snoop are explained below. The contexts are defined using the notion of initiator and terminator events. An initiator event initiates or starts the

detection of a composite event and a terminator event completes the detection of the composite event. For example, for the sequence event  $E_1;E_2$ ,  $E_1$  will be the initiator event and  $E_2$  will be the terminator event.

- **Recent:** In the recent context, only the most recent occurrence of the initiator for any event (primitive or composite) that has started the detection of that event is used. When the event occurs, the event is detected and all the occurrences of events that cannot be the initiators of that event in the future are deleted. In this context, not all occurrences of a constituent event will be used in detecting a composite event. Furthermore, an initiator of an event will continue to initiate new event occurrences until a new initiator occurs.
- **Chronicle:** In the chronicle context, the initiator-terminator pair is unique for an event occurrence. The oldest initiator is paired with the oldest terminator for each event (i.e., in chronological order of occurrence). In this context, the same primitive event occurrence is used at most once for computing the parameters of the composite event.
- **Continuous:** In the continuous context, each initiator of an event starts the detection of that event. A terminator event occurrence may detect one or more occurrences of the same event. The initiator and terminator are discarded after an event is detected. There is a subtle difference between the chronicle and the continuous contexts. In the former, pairing of the initiator is with a unique terminator of the event whereas in the latter multiple initiators can be paired with a single terminator of that event.
- **Cumulative:** In the cumulative context, all occurrences of an event type are accumulated as instances of that event until the event is detected. Whenever an event is detected, all the occurrences that are used for detecting that event are deleted. Unlike the continuous context, an event occurrence does not participate in two distinct occurrences of the same event in the cumulative context.

As mentioned earlier, rules can be defined both on primitive and composite events. A rule consists of a condition and an action. A condition is a Boolean entity that either returns true or false and an action is simply a set of statements. When an event occurs, the action is executed only if the condition evaluates to true. A rule can be specified with a coupling mode, a triggering mode and a priority. The coming sections describe these attributes of a rule.

## **2.4. Coupling Modes**

Coupling modes as described in HiPAC [1], specify when a rule is to be executed relative to the event firing the rule. They were initially proposed for a transaction based execution environment such as a DBMS. In a transaction based execution environment, all the events occur within some transaction. The coupling mode of the rule indicates when a rule should be executed relative to the event occurring in the triggering transaction. HiPAC defines three coupling modes, namely the immediate, deferred and the detached modes. The three coupling modes are briefly explained below. It should be noted that the execution environment of object-oriented applications, the environment for which active capability is being proposed in this report, is not transaction-based. In the definitions below, the difference in the meaning of the coupling modes between a transaction based

environment and a non-transaction based environment (the current environment) is also explained.

- **Immediate:** In the immediate coupling mode, the fired rule is executed immediately after the event is detected, in which case the execution of the triggering transaction is suspended until the rule is executed. In a non-transaction-based environment, the execution of the application that raises the event is suspended until the rule is executed.
- **Deferred:** In the deferred mode, the execution of a fired rule is deferred to the end of the transaction i.e., the rule is executed just before the triggering transaction commits. However, there are no transaction boundaries in a non-transaction-based environment. For this purpose, we define two events named `executeDeferredRules` and `discardDeferredRules`. All the deferred rules are accumulated from the beginning of the application and executed when the `executeDeferredRules` event is explicitly raised by the application. The application can also raise the `discardDeferredRules` event at which point all the deferred rules accumulated thus far are discarded. It is implicit that whenever a deferred rule is fired, it is accumulated. The accumulated rules are either executed or discarded depending on one of the above events explicitly raised by the application. Thus, unlike a transaction-based environment where the deferred rules fired in a transaction are executed at transaction commit time; deferred rules are executed by an explicit event raised by the application, in our case.
- **Detached:** In the detached mode, the rule is executed in a separate transaction but after the triggering transaction has committed. Note that in the deferred mode, although the rule is executed at transaction commit time, it is executed in the same transaction as the triggering transaction. On the other hand, a rule in the detached mode is executed in a separate transaction from the triggering transaction, after the triggering transaction commits. Since there are no transactions in the current environment, the detached mode is not supported in the current implementation.

Although the above explanation speaks about the coupling between an event and a rule, there actually exists a coupling between the event and the condition as well as the condition and the action. The coupling between an event and the condition indicates when the condition is evaluated relative to the event and the coupling between the condition and the action indicates when the action is executed relative to the condition evaluation. However, our implementation does not currently support the coupling between the condition and the action. The condition and the action of a rule are always executed together. The current implementation only supports the coupling between the event and the rule. The next section describes another attribute of a rule definition, namely the triggering mode of a rule.

## **2.5. Trigger Modes**

In Sentinel, event and rule definitions can be placed anywhere within the application. It should be noted that only named events could be used in rule definitions. Intermediate event expressions that are not named cannot be associated with a rule. The definition of an event that is used in a rule definition precedes the definition of the rule. As a result, it is possible that event occurrences that temporally precede the rule definition time itself

trigger a rule. As this might not be desirable in all situations, there is an option (the rule trigger-mode) for specifying the time from which event occurrences to be considered for the rule. Two options, NOW (start detecting all component events starting from this time instant) and PREVIOUS (all component events since the event was detected last are acceptable) are supported as rule-triggered modes, with NOW being the default. It should also be mentioned that an event is detected only if there are rules defined on that event. The way the current implementation supports these two rule triggering modes is explained below.

When a rule is defined, the value of the event occurrence counter (a long integer variable) at the time of rule definition is noted and the information is stored along with the rule. This serves as the timestamp of the rule. Whenever a rule is being fired, it is checked whether the triggering mode specified in the definition of this rule is NOW or PREVIOUS. If the rule trigger mode is NOW, it is checked if the timestamp of the rule is greater than the timestamps of all the component events that participated in the detection of the current event. If the timestamp of the rule is greater than the timestamps of all the component events, it means that all the component events have occurred after the rule definition time and the rule with triggering mode NOW is fired. Otherwise, the rule is not fired even though the event is detected. On the other hand, if the rule-triggering mode is PREVIOUS, the rule is fired without making the above check. This is because the component events that are detected both prior to the rule definition and after the rule definition time can be used in rule execution in the PREVIOUS rule trigger mode.

## **2.6. Rule Priority**

In addition to the parameter context, coupling mode and trigger mode associated with a rule, there is also a priority assigned to each rule. The default priority of a rule is a priority of 1. The priorities increase with the increase in numerical values i.e., 2 is a higher priority than 1, 3 is a higher priority than 2 and so on. Rules of the same priority are executed concurrently and rules of a higher priority are always executed before rules of a lower priority. It is possible that a rule raises events that in turn could fire more rules and so on. This results in a cascaded rule execution. Furthermore, rules can be specified either in the immediate coupling mode or the deferred coupling mode. Both the priority and coupling mode of a rule have to be taken into account for scheduling the rule for execution. The scheduling of rules based on their priority and coupling modes is described in Section 5.

## **3. OVERVIEW OF RELATED WORK**

This section discusses the other works in the field of computer science that focused on active capability in object-oriented database systems (OODBMS) as well as other object-oriented systems.

### **3.1. Ode**

Ode [10] is a database system that is based on the object-oriented paradigm. Active behavior in Ode is presented by incorporating *constraints* and *triggers* [11] without the notion of ECA rules. Both constraints and triggers consist of a condition and an action.

Constraints and triggers are defined declaratively within a class definition. Constraints are applicable to all instances of a class in which they are declared, as well as its subclasses. They are used to maintain the consistency of an object. Triggers on the other hand, are used for purposes other than object consistency and are applicable only to the instances of the class in which they are declared. The user specifies these explicitly. A condition  $C_i$  is paired with an action  $A_i$  only, forming a constraint or trigger. Constraints and triggers are fired as a result of the invocation of any non-constant member function. Thus events in Ode are considered as the disjunction of all non-constant member functions. A non-constant member function changes the values of the attributes of a class whereas a constant member function does not. Events are generated as a result of the invocation of non-constant public member functions. Private and protected member functions do not generate events. All events signaled by an object of class 'A' cause the evaluation of all constraints and triggers declared within class 'A'. Constraints and triggers are precompiled into each place in the code where they might be activated, specifically, at the end of each non-constant public member function and before the commit of every transaction.

Ode has also proposed a language for specifying composite events. Basic (primitive) events are defined and composite events are constructed by applying operators to basic events. The basic events that are supported are object state events (creation, deletion, access, update and read) method execution events, (before or after the execution of a method) timed events and transaction events. The event operators supported are *relative*, *prior*, *sequence*, *choose*, *every*, *fa* and *faAbs*. An event occurrence is represented as tuple of the form  $\langle \text{primitive event}, \text{event identifier} \rangle$ . An example of an event identifier is defined as the time at which the primitive event occurred. Basic events can be qualified with a mask, thus producing logical events. A mask is an optional predicate that allows users to specify more complex events. In Ode, detection of composite events is accomplished by using finite automata. Each event expression has an automaton associated with it that reaches the acceptance state when the event is raised. Input to the automaton is the event history and the sequence of logical events of the object with which the automaton is associated. An event history is defined as a set of event occurrences with no two-event occurrences having the same event identifier.

Events in Ode are treated as expressions declared within class definitions at compile time. This approach has several disadvantages. First, the treatment of events as expressions results in a dichotomy between events and other objects. Second, events cannot be created, deleted and modified dynamically. In addition, the introduction of new event types, attributes and operations requires major modifications, thus compromising the system's extensibility. The major disadvantage of this approach is the inability to express complex events that are raised by occurrences of events in different classes. In other words, Ode adopts a local view of complex events – a complex event defined in a class can only be raised by events occurring in the same class.

### 3.2. ADAM

ADAM [12] is an active OODB implemented in PROLOG. It adopts the ECA format for specifying rules and treats rules in the same way as other objects in the system. Thus it provides a uniform treatment to rules and other application objects. Within the rule object, rule operations are implemented as methods. ADAM supports *database events*,



*clock events* and *application events*. Events in ADAM are generated either *before* or *after* the execution of a method. In order to create an event, the user must specify the *name of the method* generating the event and *when* the event should be raised. Events are also treated as objects that are created, modified and deleted in the same fashion as other objects. When an event is raised, all the method's arguments are passed by the system to the condition and action parts of the rule. Thus, the condition and action code may use the method arguments during their execution. ADAM does not support complex events due to its treatment of events as objects. ADAM supports only the immediate coupling mode and does not support the other coupling modes proposed in HiPAC.

As mentioned earlier, rules in ADAM are treated as objects. An object's definition is enlarged to indicate which rules to check when the object raises an event. Inheritance of rules from super classes to subclasses is supported. However, the way by which inheritance is supported is specific to PROLOG and hence cannot be easily applied to other object-oriented programming languages. A rule defined in a class is applicable to all the instances of the class. ADAM does not efficiently allow a rule to be applicable to only one instance of a class. This is accomplished by disabling the rule for all other instances. ADAM allows the modification of condition and action parts of the rule dynamically at run time. Thus, rule implementation could be either provided either at compile time or run time. The interpretive environment in which ADAM is implemented causes the dynamic characteristics provided by ADAM. Our implementation also allows dynamic rule modification. Although Java is not fully interpretive, it is possible to modify the rule implementation at run time. This is due to the dynamic class-loading feature available in Java.

### 3.3. Samos

Samos [13, 14] addresses event specification and detection in the context of active object-oriented databases. Although there are some differences between Snoop and Samos in the event specification languages, they differ primarily in the mechanism for event detection. Samos uses modified colored Petri nets called Samos Petri Nets to allow flow of information about the event parameters as well as the event occurrence.

Petri nets are not as efficient as an event graph for detecting composite events. This can be explained by a small example. Consider three primitive events E1, E2 and E3 and two composite events C1 (E1; E2) and C2 (E1, E3). Note that E1 appears both in C1 and C2. To combine the Petri Nets for the two composite events, E1 has to be duplicated into E1' and E1''. This results in duplicating Petri nets equal to the number of common event expressions that E1 participates in. Duplication of events leads to unnecessary and excessive storage requirements. In contrast, the event graph does not duplicate events and thus offers a more optimal usage of memory.

Also, Samos detects events only in the chronicle context. But, event detection in the other contexts too may be useful for some applications. In our implementation, the semantics of contexts is built into the operator nodes. Thus, a single instance of the event graph detects events in all the four contexts. Samos has to generate a different Petri Net for each context. Again, this demands very high storage requirements and the processing could also be slower because of the high storage needed. Also, we can generate the event graph as and when the event expression is specified and the context information could be added

later. But Petri Nets need to be built again if the context information is not specified beforehand but added afterwards.

Samos uses the layered approach for providing active capability. In a layered approach the underlying DBMS is augmented with a layer that is responsible for providing active capability. There may be some limitations on the class of ECA rules that can be supported using this approach. For example, immediate coupling mode may not be possible as the layer may not be able to suspend a transaction that is being executed by the underlying DBMS. Also, explicit and other temporal events cannot be supported in this approach without resorting to polling. One last mention about Samos is that it addresses the issue of composite event detection and rule management but does not discuss the issues of rule execution.

### **3.4. Vitria BusinessWare Process Automator**

The Process Automator in Vitria BusinessWare [17] provides a modeling environment that captures business objects, events, rules and processes and uses them to build a collection of business policies. It incorporates four modeling techniques into an integrated process-development environment: business object models, business event models, business process (or "state") models and business rules. It uses a graphical modeling language to create process charts that describe the different stages of a business process. Process charts use graphical elements to describe processes in terms of discrete states and the conditions that cause the business object to move through various stages of the process by means of transition between states. Business rules and policies are expressed in event-condition-action sequences. After model states and transitions are defined in the process chart, the business rules for the process are defined using a forms-based rule editor. The rule editor is used to specify the processing condition and action when a transition occurs between the process states. Thus, an event is a transition from one state to the other. A rule condition compares a process variable with some value and such relational comparisons can be connected through logical operators. The rule action invokes methods on process objects. It has no support for composite events and context based event detection unlike our system. Also, rules cannot be specified with a priority. Rule conditions cannot perform complex computations unlike in our system. Since conditions are implemented as methods in our system, they can perform any operations that are permissible in a Java method.

### **3.5. WebLogic Events**

WebLogic Events [18] is an event notification and management service from WebLogic. It provides event registration and notification between applications across a network. It has a server that stores the event registrations from any application across the network. Event registrations are stored in a Topic Tree, which is a hierarchical, n-ary tree of period-separated words, where each word represents a node at a particular level in the tree. Each level in the hierarchy represents a greater level of specificity. The whole event service flows through the Topic Tree. When an application publishes an event, it is propagated down through the topic tree until it reaches the registration corresponding to the topic. When the event matches a registration, the registration's 'evaluate' method is called, and if the evaluation succeeds, the 'action' method of the registration is called. The 'evaluate' and 'action' methods associated with each registration correspond to the

condition and action of a rule. Since a registration is associated with a single evaluate and action pair, only a single rule can be defined on an event whereas our system allows for multiple rules to be defined on an event. The fundamental operations that can be performed on the topic tree are listed below:

- EventRegistration operations - An application registers interest in a particular topic (a node in the topic tree), by sending an EventRegistration.
- EventMessage operations – Any application on the network can generate an event for a particular topic, by sending an EventMessage.

The topic tree is dynamically built inside the WebLogic server as clients subscribe to event topics. If a client subscribes to an event topic that does not exist in the topic tree, a new node is created corresponding to the event. It can be seen that only the leaf nodes in the topic tree correspond to events and all the intermediate nodes are merely used as a branching condition. All the leaf nodes correspond to primitive events and there is no mechanism to combine the primitive events to form composite events.

The WebLogic Event Server provides communications between applications via a subscribe and publish paradigm. When a client subscribes to an event on a WebLogic server, it can specify extra conditions that must be satisfied before an event is forwarded to it. This is achieved via an evaluator that runs on the server. The evaluator checks the conditions specified by the client before forwarding the event to the client. The event action () mentioned above executes either at the server or the client. WebLogic also supports event parameters as a set of name-value pairs that further qualify the event. An event is submitted to the server with a set of event parameters.

### **3.6. Sentinel**

Sentinel [4, 5, 19, 20, 21] is an integrated active DBMS incorporating ECA rules using the Open OODB Toolkit from Texas Instruments. Event and rule specifications are seamlessly incorporated into the C++ language. Any method of an object class is a potential primitive event. The event occurs either at the beginning of the method or at the end of the method. Composite events are defined by applying a set of operators to primitive events and/or composite events. Events and rules are specified in a class definition. In addition, Sentinel supports events and rules that are applicable to a specific object instance alone. In that case, events and rules are specified within the program where the instance variables are declared. This ability to declare events and rules outside of a class allows for composing events across classes. It can be recalled that a significant drawback of Ode was that a composite event could only be composed of events within the same class but not from different classes. This is because Ode does not support event definitions outside the class. Sentinel overcomes this drawback by allowing event definitions outside the classes too.

The parameters of a primitive event correspond to the parameters of the method declared as the primitive event and other attributes, such as the time of occurrence of the event. The processing of a composite event involves not only its detection, but also the computation of the parameters associated with the composite event. The parameters of the event (primitive or composite) are passed onto condition and action portions of a rule. The parameters associated with the detection of an event can be different in different

contexts. Sentinel supports all the four-parameter contexts specified in HiPAC namely, *recent*, *chronicle*, *continuous* and *cumulative* contexts.

An event can trigger several rules, and rule actions may raise events that can trigger other rules. Sentinel supports multiple rule executions, nested rules executions as well as prioritized rule executions. Out of the three coupling modes (immediate, deferred and detached) specified in HiPAC, Sentinel currently supports immediate and deferred modes of rule execution.

#### **4. DESIGN OF THE EVENT DETECTOR**

This section describes the design of the event detector in Java. First, it mentions the functionality that is to be supported by the event detector and then it describes how the event detector has been designed to accomplish this functionality. This section also describes the main classes used in the implementation and the hierarchy among these classes. It also discusses the differences between C++ and Java approaches to implement active capability in an object-oriented environment.

The event detector should provide API to user applications for defining primitive and composite events as well as to define rules on the events. Besides, the event detector should also contain the detection logic for detecting composite events that are defined using any of the snoop event operators described in Section 2. Finally, when the events are detected, the rules defined on those events should be executed based on their coupling mode and priority. The next section describes the types of events that can be defined and are detected by the event detector.

##### **4.1. Types of Events**

As defined earlier, an event is an instantaneous and atomic occurrence. Further, events could be either primitive or composite. A Java application is a collection of classes and each class is a collection of attributes and methods. The application logic mostly consists of invoking methods on objects of these classes. For this reason, method invocations are treated as primitive events. Since an event is instantaneous, the primitive event is specified to occur either at the beginning of the method or at the end of the method. It should be noted that all methods are potential primitive events. When a method of a class is defined as a primitive event, an event occurs when any instance of the class invokes the method. But in some cases, the user may be interested in an event only when a particular instance or instances of the class invoke the method. In this case, the event should be detected only when certain instances invoke the method, but not all instances. To accommodate this, the notion of *class level* and *instance level* events is introduced. A method can be defined either as a class level or an instance level event. If the method is defined as a class level event, an event is detected when any instance of the class invokes the method. If the method is defined as an instance level event, the event is detected only when a particular instance of the class invokes the method. The instance is specified in the definition of the instance level event. A class level primitive event that is specified using snoop syntax is shown below.

```
event begin (setPriceBegin) void setPrice(float price)
```

The above denotes a primitive event named 'setPriceBegin' that occurs at the beginning of the setPrice method. Thus, the definition of a primitive event includes the name of the event, the signature of the method and the event modifier (begin or end). All primitive events should be named since they can be used in a composite event expression. An instance level primitive event in snoop syntax is shown below.

event end (setPriceIBMEnd:IBM) void setPrice(float price)

Here, IBM is the name of an instance of a class in which the setPrice method is defined. This event occurs only when the IBM instance invokes the setPrice method. Of course, the instance should be defined before defining the instance level event. It should be noted that class level events can be specified within the class definition but instance level events can be specified only where the instance is declared, either in the class or in the application.

Composite events are defined by applying one or more snoop operators to the primitive events and other composite events already defined. A composite event defined using snoop operators described in Section 2 can be defined as follows:

event or Event = (e1;e2) | (e3  $\wedge$  e4)

In addition, the application can also define temporal events that are detected at a specific clock tick. Temporal event definitions have time expressions instead of method signatures and event modifiers. There are two types of temporal events – *absolute* and *relative*. Absolute events occur at a specific point in time. For example, an event that is specified to occur on August 11, 1999 at 11:30:45 (during the last total solar eclipse of the millennium) is an absolute temporal event. The definition of this absolute event looks as follows:

event eAbs = [11:30:45/08/11/1999]

A relative temporal event is defined by specifying a relative time expression in the definition of a composite event using one of P, P\* or PLUS operators. The snoop expressions of relative temporal events look as follows:

event ePlus = setPriceBegin + [5 sec]

event ePeriodic = P(setPriceIBMEnd, [ 5 min 10 sec], buyStockIBMBegin)

The events specified in the above expressions can be either primitive or composite. Apart from these, there are external events too as mentioned in Section 1. External events occur outside the application and these occurrences are signaled to the event detector by some means that is specific to the domain of the event. For example, when a radar detects an unidentified object, it may be signaled to the event detector by some kind of update made to the application. To raise this event from the application, it has to be associated with an event signature. It may be recalled that all method events have their method signature as the event signature. The purpose of this event signature is to associate a unique string with an event. When an event is raised, its signature is used to uniquely identify the event. To facilitate the raising of external events from the application, they can be associated with a unique string that is specified at the time of event definition. The same string is used while raising the event, in order to uniquely identify the event.

It should be noted that every event is associated with a set of parameters that may be used in the execution of rules defined over the event. The next section discusses the types of event parameters that can be supported in the Java environment.

#### **4.2. Event Parameters**

It is necessary to associate an event with a set of parameters that are passed to the rule. This is because the rule should take the appropriate action depending upon the values of these parameters at the time the event occurred. Typically, the rule condition checks the values of one or more parameters of the event to see if the values satisfy a particular condition. The action part of the rule is executed only if the condition returns true. Since events are defined as method invocations, the arguments that are passed to a method are treated as the parameters for that event. Primitive data types (int, float etc), object data types defined by Java (String, Integer etc) and user defined data types can all be passed as arguments to a method. Therefore, all these types should be supported as event parameters.

There should be a mechanism for collecting the parameters of an event, storing them and then retrieving individual parameters from the parameter set. In Java, there is a generic Object data type that can contain a reference to an instance of any class type. This data type is used to store all the parameters in the parameter list. Primitive data types are stored in the parameter list after converting them to their object equivalents. During parameter retrieval, the primitive value stored in the object is returned. Class data types are stored as generic objects and they are returned as is during parameter retrieval. It is the responsibility of the user to cast the object to the appropriate type after retrieving it from the parameter list. In C++, there is no such generic data type that can store both primitive data types and pointers. Hence only primitive data types could be supported as event parameters.

A primitive event is associated with a single set of parameters whereas a composite event is associated with multiple sets of parameters, that is, the collection of the parameter sets of all the constituent primitive events. These parameters are stored in the event handle and are passed to the rule, where the values of the parameters can be retrieved and used. The next section on implementation describes the data structures used for storing the parameters and the way parameters are inserted and retrieved from an event handle. The next section describes how a rule is modeled based on the ECA rule paradigm.

#### **4.3. Rules**

A rule as mentioned earlier, consists of a condition and an action. A condition is a function that returns a Boolean value. Since all execution in a Java application is through methods defined in a class, rule condition and action are defined as methods associated with some class. There is one important capability required for defining conditions and actions as methods. It should be noted that the condition and action are pieces of code to be executed. When the rule is fired, the application should be able to jump to the place where the condition and action codes are located, and execute them. This is possible if there is a reference that can point to the appropriate pieces of code. C++ does not support references to methods declared in a class but there can be references possible to C functions via function pointers. For this reason, conditions and actions are implemented

as functions in the C++ implementation. But Java supports references to class methods and hence conditions and actions can be implemented as methods in Java. If the condition and action are implemented as methods of a class, all the attributes of the class can be used in condition checking as well as action execution. This is not possible if conditions and actions are implemented as functions.

Java reflections are used to obtain the references to the condition and action methods defined in a class. Unlike other methods, condition and action methods cannot be passed any number of arguments. Only a fixed number and type of arguments can be passed to these methods. This is because the arguments passed to a condition and action method should be known to the event detector in order to obtain references to these methods using Java reflections, at run time. Therefore, all condition and action methods take a single argument of type `ListOfParameterLists`. This data type is defined in the event detector, which stores a list of parameter lists. A parameter list is a set of parameters associated with a primitive event. The following code segment shows how a reference to a method is obtained.

```
Class[] methodParams = new Class[1];
methodParams[0] = Class.forName("Sentinel.ListOfParameterLists");
Class classObj = Class.forName("Stock");
Method methodObj = classObj.getDeclaredMethod("checkPrice", methodParams);
```

In addition to a condition and an action, there are other attributes associated with a rule that are described in Section 2. These are the coupling mode, trigger mode and the priority. As mentioned in Section 2, composite events can be detected in different contexts. If the user wants a composite event to be detected in two different contexts and define rules on them, he has to specify the same event expression in two event definitions (one for each context) and define rules on each of them. This results in two event definitions although both of them contain the same event expression. This can be avoided if the context information is specified as part the rule definition, instead of specifying it as part of the event definition. Now, the composite event is defined only once and the rule defined in a particular context is fired only if the event is detected in the same context. Thus, the parameter context is also specified in the definition of a rule. The snoop expression for a rule definition is shown as:

```
rule r1[setPriceEnd, checkPrice, buyStock, RECENT, IMMEDIATE, NOW, 5]
```

where 'checkPrice' and 'buyStock' are the names of the condition and action methods, respectively, and the rule is defined in the immediate coupling mode, with trigger mode now and a priority of 5. The rule is fired only when event 'setPriceEnd' is detected in the recent context. It should be noted that the above attributes of the rules are optional parameters. If the user does not specify any of the parameters, the default values will be associated with the rule. The default context is the recent context, the default-coupling mode is immediate, the default trigger mode is now and the default priority is 1.

In addition to adding rules for an event, a rule can also be deleted at any time within the application. It is also possible to disable a rule at any time and enable it again. When a rule is deleted, the rule is removed and is no longer available to the application. If the application wants the rule at a later point, the rule has to be created again. On the other

hand, when a rule is disabled, it is marked disabled but not removed. The application can enable it again at a later point without having to create the rule again.

#### **4.4. Instance Level Rules**

As mentioned in section 4.1, it is possible to define instance level events. The definition of an instance level event takes the name of the instance as one of its arguments. These events are named and can be used in composite event expressions. In many applications, a number of rules are defined on instance level events. Rules defined on instance level events are called instance level rules. Defining an instance level rule typically involves two steps: (1) defining the instance level event and (2) defining the rule on the instance level event. These two steps are shown below:

```
event begin(setPriceIBMEnd:IBM) void setPrice(float price)
rule r1[setPriceIBMEnd, checkPrice, buyStock]
```

Most of these instance level events are not used in composite event expressions. As will be explained section 4.6, all events are represented as an event node in an event graph. It is not necessary to create separate primitive event nodes for these instance level events if they are not used in composition. Thus, in order to combine the above two steps of defining an instance level rule into one step, another form of rule definition is introduced which is shown as:

```
rule r1[setPriceEnd, IBM, checkPrice, buyStock]
```

Here, the name of the instance is specified in the rule definition. It is not required to define the instance level event on IBM instance in this case. 'setPriceEnd' is a class level event whose definition looks like:

```
event end(setPriceEnd) void setPrice(float price)
```

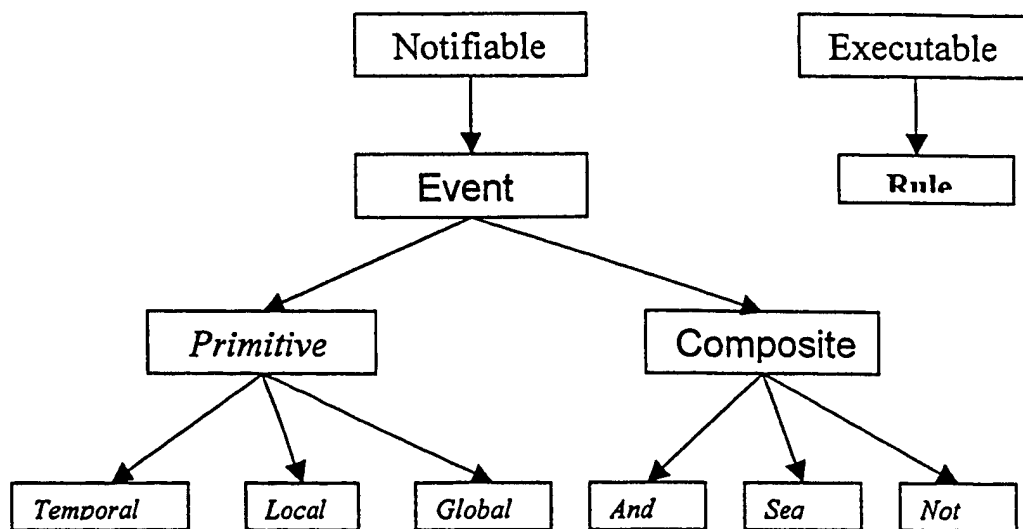
In order to accommodate instance level rules, a primitive event node contains a special data structure. This will be explained in section 4.6.1. The next section describes the important classes used in the implementation and the hierarchy among those classes.

#### **4.5. Class Hierarchy**

Figure 1 depicts the hierarchy among the key classes used in the implementation. There is one class for each event operator and a class for a primitive event. A primitive event can be either a temporal event, a local event (event detected within the application) or a global event (event detected outside the application). There is also a rule class that is used to instantiate rule objects.

In Figure 1 different fonts depict different kinds of classes. The classes Event and Composite are abstract classes, the classes Notifiable and Executable are interface classes and all the other class names in italics are normal classes. An abstract class is a class that can only be sub-classed but cannot be instantiated. For example, the Event class is an abstract class that can only be sub-classed but cannot be instantiated. This is because although there are instances of primitive, temporal, and composite events (AND and SEQ for example), there is no need to instantiate an event itself.





**Figure 1** Class Hierarchy

An interface is a collection of method definitions (without implementations) that indicate a certain behavior of the classes that implement the interface. The classes that implement an interface provide implementations for the methods in the interface. For example, the Notifiable interface has a notifyEvent method that is implemented by all the operator classes. The notifyEvent method implements the detection logic in an operator class according to the semantics of the operator. The classes AND, SEQ, and NOT are normal classes whose instances represent the composite event nodes in the event graph. The event graph is described in the next section.

#### **4.6. Event Graph**

The application defines a set of primitive events, composite events and rules on events (both primitive and composite). The relationship between these entities is established by means of a subscription and notification mechanism that is represented by means of an event graph. An event graph is a graph whose nodes are the primitive events, composite events and rules defined in the application. A composite event subscribes to all its constituent primitive events and the composite event is notified whenever the constituent primitive events are detected. Similarly, rules are also subscribed to events and are notified whenever the events are detected.

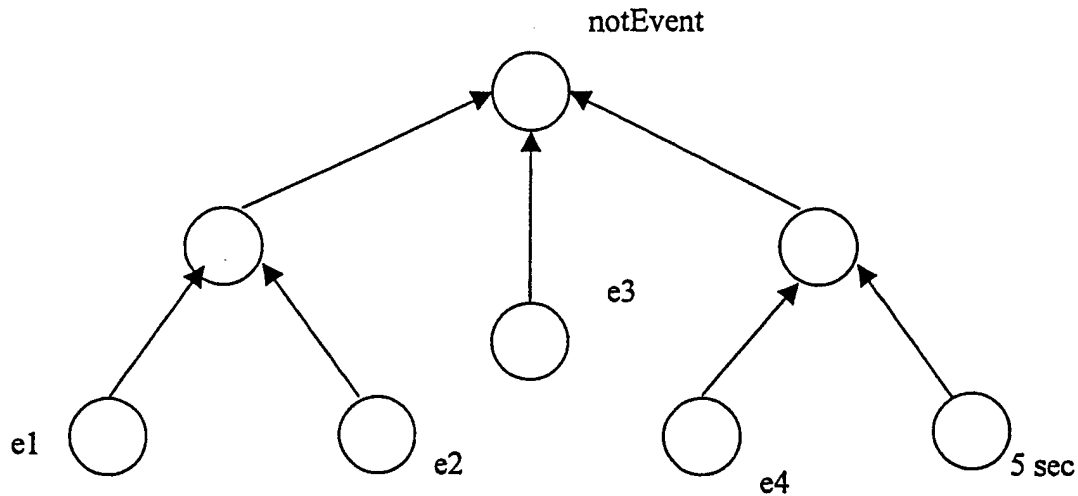


Figure 2 An Example Event Graph

Figure 2 depicts an example event graph for the following composite event:

event notEvent = NOT((e1  $\wedge$  e2), e3, (e4 + [5 sec]))

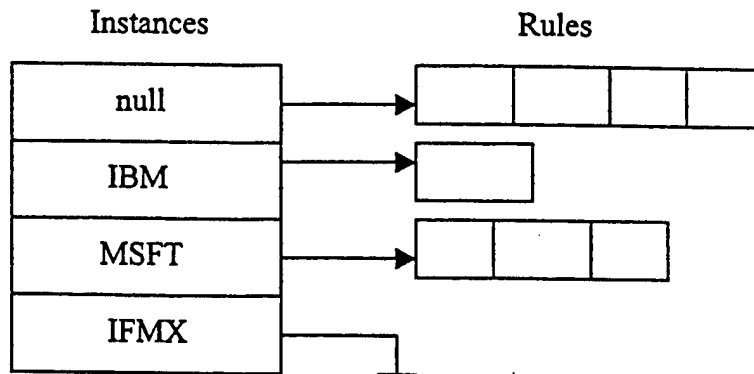
Each event is represented as an event node in the graph, and the event nodes are connected by their subscription relationships. An internal node of the event graph represents a composite event whereas a leaf node represents a primitive event. Every node in the event graph has a list of event subscribers and a list of rule subscribers. The list of rule subscribers contains references to the rule objects that denote the rules defined for that event. The list of event subscribers contains references to the event nodes of those composite events that have subscribed to the event. A composite event subscribes to the event nodes of all its constituent events.

#### 4.6.1 Primitive Event Node

A primitive event node is a leaf node of the event graph that denotes a primitive event. The primitive event can be either *local* or *global*. Local events occur within the application and global events are external events that occur outside the application. A primitive event node is created for every primitive event (class level or instance level) defined in the application. It contains the name of the primitive event, the method signature and the event modifier specified in the primitive event definition. Apart from these, in order to accommodate the instance level rules described in section 4.4, a primitive event node contains an instance-rule list that contains a list of instances and a list of rules associated with each instance.

As shown in Figure 3, class level rules are associated with a null instance and instance level rules are associated with the instance specified in the definition of the instance level rule, as mentioned in section 4.4. Storing all the instance level rules in the same primitive event node would reduce the overhead of creating multiple event nodes. The event nodes are not necessary if the corresponding events are used only for defining rules but not in any composite event expression.

Separate primitive event nodes are created for those instance level events that are explicitly defined by the user application. This is because these events can be used in composite event expressions. In the nodes for these events, the instance-rule list contains only a single instance and the set of rules defined on this instance level event.



**Figure 3 An Instance list associated with a primitive event**

#### 4.6.2 Composite Event Node

A composite event consists of two or more primitive events. The composite event node stores the name of the composite event, the references to the nodes of its constituent events and the parameters of all the constituent events. Both primitive and composite event nodes also store the list of subscribed events and the list of rules defined over that event. The parameters of a constituent event are stored in a data structure called the event table. The event table is described in the next subsection.

#### 4.6.3 Event Table

Figure 4 shows an event table. An event table consists of a set of event entries. Each entry in an event table denotes an event occurrence. An event entry consists of a set of primitive events (that is, a list of parameter lists) and four bits. It should be noted that the constituent events of a composite event can be composite events themselves. As already mentioned, a single parameter list denotes a primitive event occurrence and a list of parameter lists denote a composite event occurrence. The four bits stored in an event entry are used to detect a composite event in different contexts. Composite event detection in different parameter contexts using these four bits is explained in the next section.

$e_1^1 e_2^1 e_3^1$	0010
$e_1^1 e_2^2 e_3^1$	0110
$e_1^1 e_2^1 e_3^2$	1110
$e_1^1 e_2^2 e_3^2$	0001

Figure 4 An Event Table

#### 4.6.4 Rule Node

The rule node stores the name of the rule, the references to the condition and action methods, the context, coupling mode, triggering mode and the priority of the rule. The rule node also stores a rule enable/disable flag that is used to indicate whether the rule is enabled or not.

### 4.7. C++ Versus Java Approach

Since Sentinel has an event detector implemented in C++, it would be worthwhile to bring out the differences between C++ and Java languages in terms of what they allowed us to do and not to do, while implementing the event detector. The following subsections illustrate the differences between the C++ and Java approaches.

#### 4.7.1 Representation of a Rule

As it is known, a rule has three components namely event, condition and an action. All the three components are specified as part of the rule definition. It should be noted that the condition and action are pieces of code to be executed. When an event occurs, the application should be able to jump to the place where the condition and action codes are located, and execute them. This is possible if there is a reference that can point to the appropriate pieces of code. C++ does not support references to methods declared in a class but there can be references possible to C functions via function pointers. For this reason, conditions and actions are implemented as functions in C++. But Java supports references to class methods and hence conditions and actions could be implemented as methods in Java. It is more useful if conditions and actions are executed as class methods than as stand-alone functions. This is because the methods are bound to the objects invoking the methods and makes it more object-oriented.

#### 4.7.2 Temporal Events

As was mentioned earlier, it is possible to access the system timer from C++ programs. It is possible to set an alarm for some amount of time and be notified later when the alarm rings. The setitimer system call is used to set the alarm. The application then catches the SIGALRM signal generated by the system timer at the end of the alarm time. The corresponding temporal event node is notified when the application catches the SIGALRM signal. It is not possible to do it the same way from Java applications since Java does not support system calls to be made to the underlying operating system. It is

also not possible to catch signals generated by the operating system from a Java application.

An alternative to detect temporal events is to use Java Native Interface (JNI) to call C functions from Java and calling back to Java from C. The C function can be used to set the timer and catch the timer interrupt from the operating system. A Java method can be called when the function catches the timer interrupt. However, there are two disadvantages to this approach. First, calling C functions from Java and calling back Java from C through JNI are both expensive operations. If this is done frequently within the application, it will severely affect the performance of the application. Second, using system calls in C would make the application platform dependent. This is because different operating systems have different system call interfaces.

Therefore a different approach has to be taken to implement temporal events. A combination of Java thread and the 'sleep' method call is used for this purpose. A Timer thread is implemented that simulates a timer to detect both absolute and relative temporal events. The way this is implemented is described in the next section on implementation.

#### 4.7.3 Vector and Hash table Data Types in Java

Java provides the Vector and Hash table data types for storing a collection of elements. A Vector is a linear list that can store heterogeneous types of elements and also provides an indexed access to its elements. It is like a growable array of objects. A Vector is more efficient than a linked list since the elements in a Vector can be accessed through an index whereas the elements in a linked list can only be accessed serially. It is particularly more efficient if there are more additions to the Vector than deletions. In the implementation, Vectors are used for storing the list of event subscribers and the list of rules at every event node. They are also used for storing the list of parameter lists at a composite event node. This is because these lists mostly involve additions and none or very few deletions.

The Hash table data type also stores a collection of elements. It is an associative array that stores a key-value pair. Both the key and the value could be any type of Java objects. A value can be stored against a key and retrieved later using the same key. It gives a faster look-up speed when compared to searching for a particular value in a linked list. It should be noted that it is not exactly equivalent to the general hash table since unlike a general hash table, it is not possible to store more than one value against the same key in a Java Hash table. In this implementation, Hash tables are used to store the mapping of event names and event nodes, event signatures and event nodes and rule names and rule nodes. This is because the references to the event nodes and rule nodes do not change and also because the references can be uniquely associated with a key such as the event name, event signature or the name of a rule.

## 5. IMPLEMENTATION

This section describes the implementation details of the event detector. First, primitive and composite event detection is described. Rule execution based on priorities and

coupling modes is also described. This section also describes how the event detector runs in a separate thread from the application.

### 5.1. *Primitive Event Detection*

This section describes how primitive events are detected in a Java application. As part of the primitive event definition, the application specifies a name for the primitive event, the fully qualified name of the class in which the method associated with the primitive event is defined, the event modifier and the complete signature of the method. The API used for defining primitive events is shown below:

```
createPrimitiveEvent("setPriceBegin","Stock",EventModifier.BEGIN,"  
                                void setPrice(float)")
```

Primitive events that belong to a class and the rules on these primitive events are either defined in a *static* block within the class or can be defined anywhere in the application. The Java Virtual Machine (JVM) loads a class whenever it comes across the name of the class in the application. At class load time, the statements enclosed within the *static* block are executed. Therefore, the definitions of primitive events and rules that are specified within a *static* block are executed when that class is loaded. When a primitive event is defined using the above API, an event handle corresponding to that event is returned. The event handle is used to store the parameters of the event as well as to signal the method invocation to the event detector. How this is done is described later in this section.

The Java application initially invokes an *initializeAgent* method that returns an instance of the 'ECAAgent' class. The application can either get a default instance or a named one by passing an optional name to the *initializeAgent* API. This name can be used to get the associated ECAAgent instance anywhere in the application by using the *getAgentInstance* API. The above API is invoked on an instance of the 'ECAAgent' class. The ECAAgent instance stores the names of all events and their associated event handles in a Hash table. The primitive event definition creates a primitive event node that forms a leaf node in the event graph described in the previous section. The event detector maintains two more hashtables – one hash table stores a mapping between event names and event nodes (*eventNamesEventNodes*) and the other hash table stores a mapping between method signatures (only for primitive events) and the event nodes (*event signature event nodes*). Inside the method that is defined as a primitive event, the user adds calls to the event detector API in order to signal the invocation of a method (a primitive event occurrence) to the event detector. First, the event handles corresponding to the primitive event are obtained using the name of the primitive event. After obtaining the event handles, the arguments of the event method are inserted into the event handles through the 'insert' API. The parameters are inserted by specifying the event handle, a symbolic name of the parameter and the parameter itself. Finally, the event handles and the instance which invokes the method (*this*) are passed through the 'raiseBeginEvent' API. For *end* events, the 'raiseEndEvent' API is used and is placed at the end of the method.

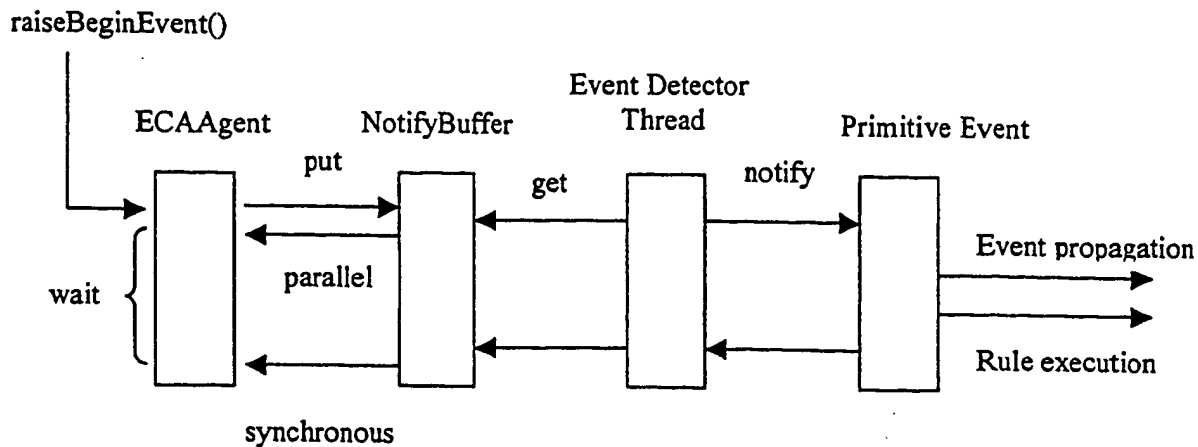
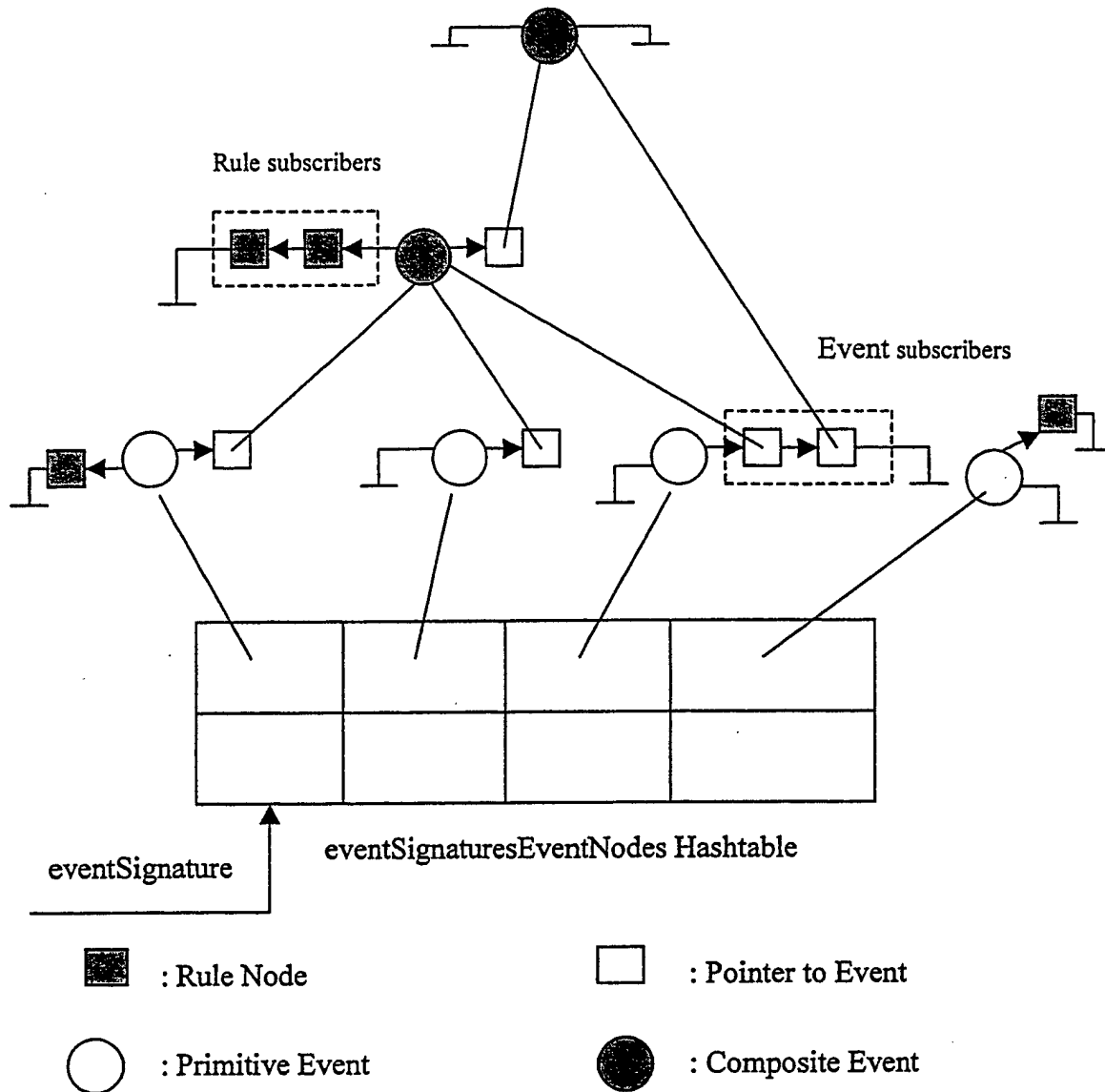


Figure 5 The Event Detector Thread

As mentioned earlier, the event handle stores the signature of the method. This signature is used to get the corresponding event node from the `eventSignaturesEventNodes` hash table mentioned above. Then, the node is notified about the occurrence of the primitive event and the parameter list stored in the event handle is passed to the node as an argument to the `'notifyEvent'` call. Thus, the invocation of an application method (begin or end) is detected as a primitive event by the event detector (Figure 5). It should be noted that there is a timestamp associated with all primitive events. Currently, the timestamp consists of an integer counter that is incremented at every primitive event occurrence. After the primitive event is detected, its parameters are propagated to all the composite events that have subscribed to it.



**Figure 6** An Event Detector Graph

Finally, the list of rule subscribers is traversed and the rules are executed according to their priority and coupling mode. Rule execution based upon rule priority and coupling mode is described in a later section. The next section describes how composite events are detected.

As described in Section 4, a primitive event node contains an instance-rule list that contains a list of instances and a set of rules associated with each instance. When a primitive event occurs and the corresponding class level node is notified, the instance rule list is traversed and checked to see if the event instance (the instance which invoked the event method) is present in the list of instances. If the event instance is present, the list of rules associated with the instance is traversed, and the rules are executed.



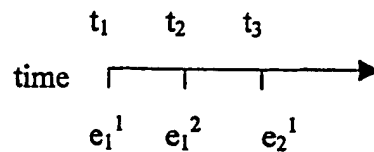
## 5.2. Composite Event Detection

A composite event is composed of two or more primitive events using one or more of the snoop operators. Snoop operators are either binary or ternary. For example, the 'AND' operator is a binary operator with two constituent events whereas the 'NOT' operator is a ternary operator with three constituent events. A composite event is associated with a list of parameter lists, i.e., the parameter lists of all the constituent primitive events. Every composite event has an initiator event that initiates the detection, and a terminator event that completes the detection of the event. The composite event is detected when the terminator event is detected. It should be noted that a composite event is detected only when either there are rules defined on that event or rules defined on another composite event for which this event is a constituent event. The composite event is to be detected and propagated in the later case because it could lead to the detection of the dependent composite event over which a rule is defined. For this purpose, there are four integers stored at each node. Each integer denotes the sum of the rules defined on that event and all the dependent events in a particular context. A composite event is detected and propagated in a context only when the corresponding integer is non-zero. Whenever a rule is defined or enabled on an event in a particular context, the corresponding integer in that node is incremented. Also, the event graph (Figure 6) is recursively traversed from that event node until the leaf nodes are reached and the corresponding integer is incremented in all the nodes encountered. When a rule is deleted or disabled, the same procedure is followed and the corresponding integer is decremented. The process of composite event detection in all four parameter contexts is described next.

As mentioned earlier, a composite event can be detected in four different contexts. To illustrate this, consider the definition of the following composite event using the AND operator:

$$\text{event andEvent} = \text{AND}(e_1, e_2)$$

Consider the event occurrences shown on the timeline below.



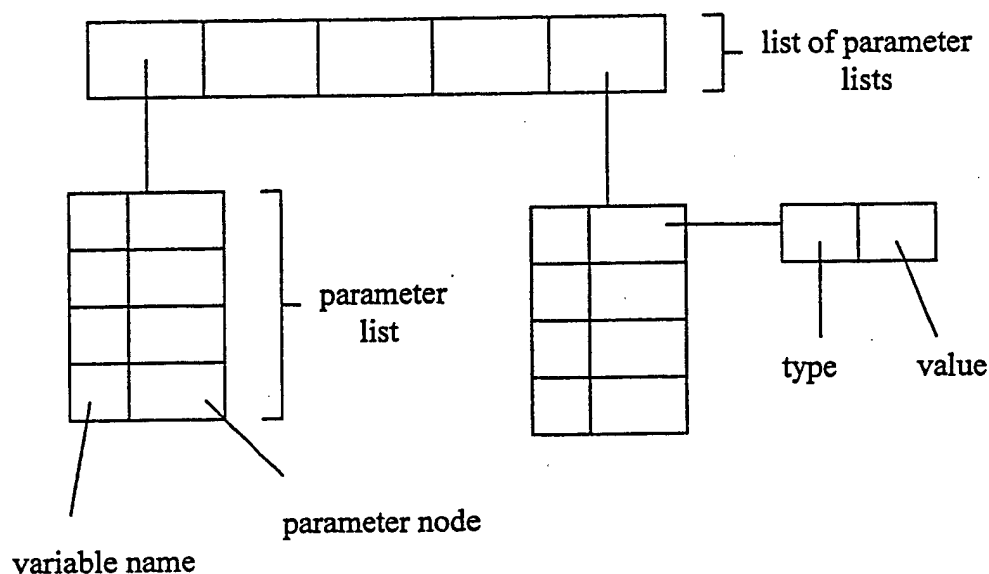
The AND event is detected when  $e_2$  occurs. It is not clear whether  $e_2^1$  should be paired with  $e_1^1$  or  $e_1^2$ . In order to avoid this disambiguate, parameter contexts were introduced. For this example,  $e_1^2$  and  $e_2^1$  are paired in the recent context whereas  $e_1^1$  and  $e_2^1$  are paired in the chronicle context. In the continuous context, two events are detected at the same time –  $e_1^1 e_2^1$  and  $e_1^2 e_2^1$ . In the cumulative context, a single event is detected with constituent events  $e_1^1 e_1^2 e_2^1$ . Thus different events are paired in different contexts at the same time. To accomplish this, the following procedure is followed for detecting composite events.

In every composite event node, there is an event table stored for each constituent event. Each entry in the event table stores an event occurrence and a set of four bits containing information about the four different contexts associated with the event. The event

occurrence could be either a single event (a single parameter list) or a set of events (a list of parameter lists). At the first level of composite event nodes in the event graph, the event occurrence is a single event. At composite event nodes higher up in the graph, the event occurrence is a set of events and the size of the set increases as we move higher in the graph. The four bits associated with an event occurrence denote the four contexts. The first bit denotes the Recent context, the second bit denotes the Chronicle context, the third bit Continuous context and the fourth bit denotes the Cumulative context. If a particular bit is set, it means that this event is yet to participate in the detection of the composite event in the corresponding context. When an event occurrence is used to detect the composite event in a particular context, the corresponding context bit is reset according to the semantics of the operator.

Once an event is detected, its parameters are propagated to all its subscriber events. It should also be noted that an event is detected in a particular context only when there are rules defined on a composite event in that context. In other words, a context bit for an event occurrence is set only when there is at least one rule defined in that context. This minimizes the propagation of events in the event graph.

### 5.3. Parameter Lists



**Figure 7** The data structure used for storing the parameter

As already mentioned, the parameters of a primitive event are the arguments to the method associated with that event. The name of a parameter is the name of the argument. If the parameters are inserted into the parameter list in the same order as they are passed to the method, the position of the parameter would be its position in the method definition. When parameters are inserted into a parameter list, they can be retrieved using either the name of the parameter or the position of the parameter in the parameter list. If the parameters are to be retrieved using their name, both the parameter and its name are to be stored whereas if they are retrieved by their position only, only the parameter needs to be stored. To allow for both kinds of retrievals, we store both the parameter and its

name in the parameter list. Since there should be an association between the parameter and its name, a hash table is used for the parameter list. This hash table is a mapping between parameter names and parameter nodes. As shown in the Figure 7, a parameter node stores the type and value of the parameter. As mentioned in Section 4, the value of the parameter is stored as an Object type. The type information is needed in order to cast the Object to the appropriate type at run time.

As already mentioned, a composite event is associated with a list of parameter lists. It should be possible to search through this list for a particular parameter list either by position or content. For this purpose, a vector is used to store the list of parameter lists. In a vector, the individual parameter lists can be accessed both by position as well as sequentially. A set of API is provided on the list of parameter lists to find a particular parameter list.

#### **5.4. Detecting Temporal Events**

For detecting temporal events, there should be a mechanism for keeping track of clock time from within the program. The timer maintained by the operating system could be used for this purpose. The important aspect of temporal event detection is that the time notifications from the operating system should be asynchronous to the execution of the program. The program should be notified at the moment the designated time expires, irrespective of the execution point in the program.

Some programming languages like C provide an interface to the operating system timer. This interface lets application developers to set an alarm for a specific period of time and then catch the signal generated by the timer at the end of the expiration of alarm time. For absolute temporal events, the alarm is set at the time of the definition of the event. For relative temporal events, the timer is set when the precedent event occurs. For achieving this capability, the application program should be able to catch signals generated by the operating system. As part of the OS independence, the Java programming language does not provide a mechanism to access the system timer and catch the signals generated by the operating system. Therefore it is not possible to set the system timer and receive notifications from it. But, Java provides a 'sleep' method call that causes the calling thread to sleep for a specified amount of time. The sleep method call uses the timer of the underlying operating system to count the time.

There is a subtle difference between the sleep method calls in C and Java. In C, the suspension time specified by the call may be less than that requested because *any* caught signal will terminate the sleep following execution of that signal's catching routine. If the calling thread has set up an alarm signal before calling sleep, the thread sleeps only until the alarm signal occurs. This may disrupt the semantics of the application if the semantics is dependent on the actual time spent by the thread in the sleep method. In the present case, it would lead to wrong semantics since we want the thread to sleep (i.e., suspend) for the fixed amount of time and then trigger the corresponding temporal event. If the thread does not sleep for the desired amount of time, it will lead to incorrect notification of the corresponding temporal event. Moreover, if two threads are sleeping within the same process, the end of the sleep in one thread would terminate the sleep in the other thread too.

In Java, no OS signals are accessible from the application program. Hence there is no possibility of another signal causing a sleep method to return prematurely. Also, unlike in C, the termination of a sleep call from one thread does not cause another sleeping thread to terminate its sleep. However, it is possible to interrupt a sleeping thread from another thread, by calling the interrupt method on the sleeping thread. This would also be beneficial to us as it would be necessary to interrupt a sleeping thread as will be explained later in this section where the implementation of the timer is described. Thus, when a Java thread calls sleep, it is either terminated at the end of the sleep period or when another thread interrupts it. Owing to the above reasons, the sleep method call is used in a Java thread to implement timer notifications in our system. A Timer thread is implemented that simulates a timer to detect both absolute and relative temporal events. The next paragraph describes the way the Timer thread simulates the timer.

The execution of the Timer thread is relatively simple. It runs in an infinite loop sleeping for a certain period of time and then sending a notification to the corresponding temporal event node at the end of the sleep time. When the application defines a temporal event (absolute or relative such as PLUS, PERIODIC and PERIODIC-STAR events), a temporal event node is created containing the time expression specified in the temporal event. For absolute temporal events, the timer thread sleeps for the difference amount of the time specified in the absolute temporal event and the current system time. For relative temporal events, the relative time expression is first converted to absolute time and the timer thread is put to sleep for a period that corresponds to the difference between this absolute time and the current system time.

Since there can be more than one temporal event specified in the application and the same clock time may raise multiple events, there should be a way to manage all the temporal events appropriately. While the timer thread is sleeping for a certain period of time, there could be another definition of a temporal event that occurs before the current sleep time. For this purpose, the timer thread should be able to be interrupted while it is sleeping, and then put to sleep again for a different amount of time. The Temporal Event Handler component of the event detector has to take care of all these nuances by managing the temporal events appropriately. The temporal event handler is described in the next section.

### **5.5. Temporal Event Handler**

The temporal event handler consists of three classes – TimeItem, TimeQueue and Timer. The TimeItem class stores a time expression and an event\_id that refers to the precedent event for relative time events. An absolute time expression has the format– hh:mm:ss/MM/dd/yyyy, where h denotes the hour, m denotes the minute, s denotes the second, M denotes the month, d denotes the day and y denotes the year when the event should occur. This implementation also supports the specification of wild cards in one or more positions of the absolute time expression given above. The two wild cards that are supported are '?' and '\*'. The wild card '?' can replace any one position in one or more fields of the time expression. For example, the time string 12:?0:00/08/15/1999 expresses the repetitive time event which occurs at 12:00:00, 12:10:00, 12:20:00 ... 12:50:00 on 08/15/1999. The wild card '\*' can replace all the positions in a particular field of the time string. For example, the time string 12:\*00/08/15/1999 expresses the repetitive time event which occurs at 12:00:00, 12:01:00, 12:02:00 ... 12:59:00 on 08/15/1999. The

implementation of repetitive temporal events follows the algorithm specified in [22]. A relative time expression appears in the definition of one of the temporal events – PLUS, PERIODIC or PERIODICSTAR. It has the format – hh 'hrs' mm 'min' ss 'sec' where the letters stand for the same things as above. The relative time is converted to an absolute time by adding the relative time to the system time when the TimeItem is instantiated.

A TimeQueue is a linked list of TimeItems maintained in the ascending order of their time values. The temporal event handler consists of two TimeQueues namely, allItems and presentItems. It should be noted that triggering of the timer could correspond to more than one event that needs to be triggered at the same time. The list of presentItems consists of all the temporal time items that are to be notified at the same time. The timer thread sleeps for the amount of time specified in the time items of the presentItems list. Initially when the lists (allItems and presentItems) are empty, the incoming time item is put in the presentItems list and the timer thread is put to sleep accordingly. When a new time item comes while the timer thread is sleeping, the time value in the new time item may be lesser, greater or equal to the time value for which the timer thread is currently sleeping. The algorithm shown below illustrates the steps taken for managing the time queues when a new time item arrives.

Let *presTime* be the time for which the timer is currently set.

Let *currTime* be the new time value to be added.

Let *tqi* be a TimeQueueItem.

If *currTime* = *presTime*

add *currTime* to the *presItems* queue

else if *currTime* > *presTime*

add *currTime* to the *allItems* queue in the appropriate position

else if *currTime* < *presTime*

add *currTime* to the *allItems* queue

Move all the *tqi*'s in the *allItems* queue that have the same expiration time, to a temporary queue

Add all the *tqi*'s in the *presItems* queue to the *allItems* queue

Move all the *tqi*'s in the temporary queue to the *presItems* queue

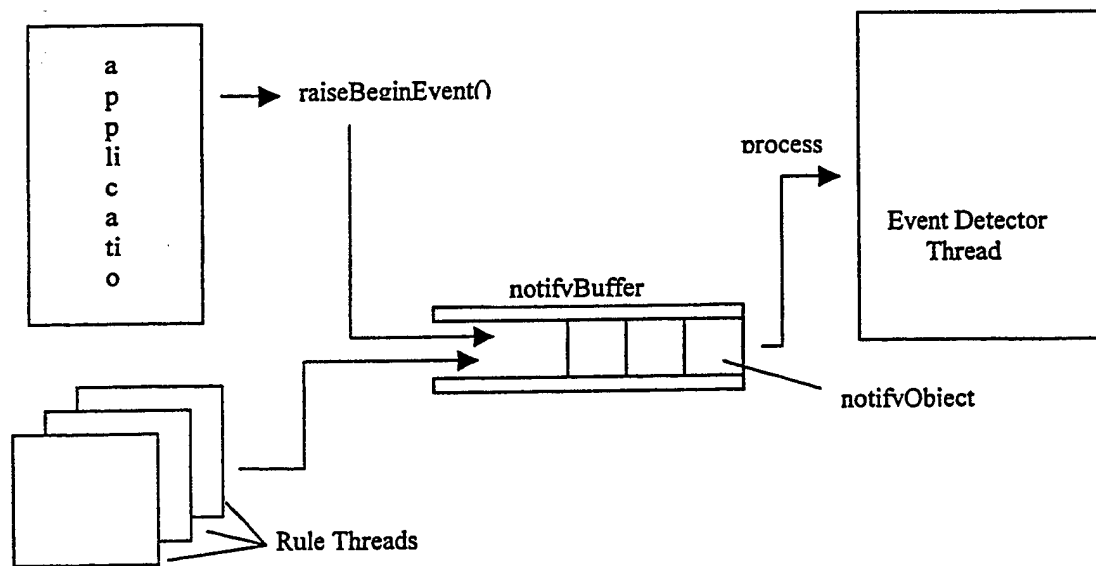
Reset the timer

In case the time value of the new time item is equal to the time value of the current time items, the new time item is added to the list of presentItems. If the new time value is lesser than the current time value, all the time items in the presentItems list are moved to the list of allItems and the new time value is placed in the presentItems list. The timer thread is interrupted and put to sleep for the new time value. On the other hand, if the new time value is greater than the current time value, it is placed in the allItems list in the appropriate position. It is to be remembered that the allItems list is maintained in the ascending order of time values. Whenever the time items in the presentItems list are processed, all the time items in the allItems list that have the same time value are moved to the list of presentItems and the timer thread is put to sleep according to the new time.

The Timer class manages the two lists mentioned above – allItems and presentItems. It extends the Java Thread class and runs in a separate thread from the application. When a new time item is added, it is put into the appropriate list and the necessary modifications are done to the two lists. This class is also responsible for calling the sleep method call with appropriate time value, interrupting the sleep if necessary and resuming the sleep again. The 'start' method for this thread is placed within a static block in the 'Timer' class. Therefore, the thread is started when the Timer class is loaded. The Timer class would be loaded when a temporal event attempts to set the timer for a time period. The timer thread runs in an infinite loop, checking if there are any time items present in the presentItems list that are to be notified.

### 5.6. Event Detector as a Thread

In the earlier implementation of the event detector in C++, the event detector runs in the same thread as the application. In other words, all the calls to the event detector were processed synchronously in the application. This implementation separates the event detector and the application into two different threads. In other words, the calls from the application to the event detector are processed in a separate thread from the application.



**Figure 8** Processing Event Notifications from Event Detector Thread

This was primarily done to have a clean separation between the application and the event detector. We believe this will help when extending the event detector to detect events across address spaces. One more advantage is that it allows the detection of events to be processed either synchronously with the application or in parallel with the application. The architecture of the event detector as a thread is shown in Figure 8.

Whenever the application makes a call to the raiseBeginEvent API, a reference to that method is constructed and placed in an object. The reference to the method is obtained by using the Java reflections.

The actual parameters used to call the method are also stored in that object. This object is then placed in a buffer called the 'notifyBuffer' that can contain more than one object at a time. Once this object is placed in the notifyBuffer, the application thread can continue and place additional objects in the notifyBuffer. The event detector runs as a thread in an infinite loop that continuously keeps getting objects from the notifyBuffer and executing the method calls stored in those objects with the parameters stored in that object.

It should be noted that only the raiseBeginEvent calls are put into the notifyBuffer and processed by the event detector thread. The event detector thread does not process the calls to the API methods for creating events and rules. This is because the API for creating events returns an event handle. Returning the event handle would not be possible if the event detector thread processes these calls, since threads cannot return a value. Therefore, the calls for creating events and rules are processed by the application thread itself and the event detector thread processes only the raiseBeginEvent calls.

Since the application thread and the event detector thread access the notifyBuffer simultaneously, the access to the notifyBuffer through its *put* and *get* methods has to be synchronized. In the implementation, these two methods are defined as synchronized methods so that only a single thread can execute one of these methods at a time. It should be noted that the application thread and the event detector thread use the same notifyBuffer object. While the application thread keeps putting objects into this buffer, the event detector thread keeps getting objects from the same buffer. The algorithms for the *put* and *get* methods are shown below.

PUT method:

Add the *notifyObject* to notifyBuffer

GET method:

If notifyBuffer is empty

wait

else return the notifyObject

RUN method of the Event Detector Thread:

while (*true*) {

Get the *notifyObject* from notifyBuffer

Process the notifyObject

if DetectionMode of event == *synchronous*

wakeup notifyBuffer

}

Application Thread:

Put the *notifyObject* into the notifyBuffer

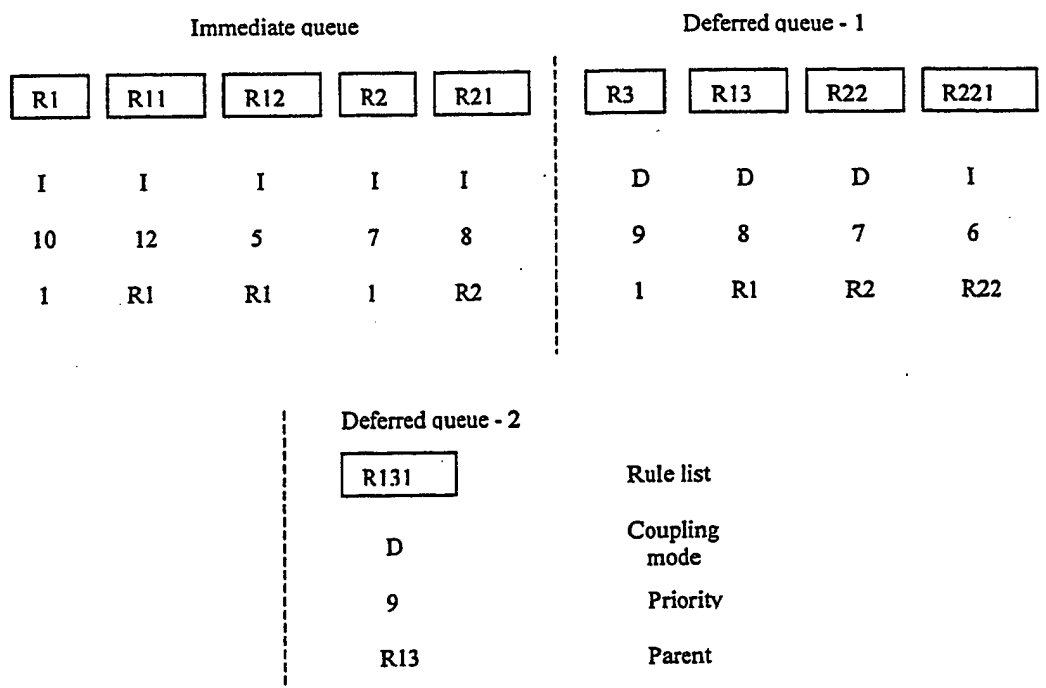
if DetectionMode == *synchronous*

wait

else return

The detection mode of the event is specified at the time of primitive event definition. The detection mode can be either synchronous or parallel. It should be noted that the detection mode is specified only for primitive events and not for composite events. This is because composite events are detected according to the detection mode of its constituent primitive events. If the event is to be detected synchronously, the application thread is made to wait until the notifyEvent call returns. On the other hand, if the event is to be detected in parallel, the application thread does not wait until the notifyEvent call returns, but continues. The information regarding the mode of event detection (synchronous or parallel) is stored in the corresponding primitive event node.

**Figure 9 Rule-list data structure**




### 5.7. Rule Scheduling

As already mentioned, when an event occurs in the application, the rules that subscribe to that event are triggered. A simplistic approach would be to let all the rules that have been triggered by an event to run one after another. This approach does not use parallelism or maximize throughput. It would be beneficial to let the rules execute in parallel, thereby reducing the overall execution time of the rules. Since each rule also has a priority associated with it, the rules are to be executed based on their relative priorities. Thus, rules with the highest priority are to be executed first and rules of lower priority are to be executed after the completion of the higher priority rules. As mentioned in Section 2, a rule is also associated with a coupling mode. Rules are to be executed based on their



coupling modes. Rules created with immediate coupling mode are to be executed at the time the event occurs whereas rules created with deferred coupling mode are executed at a later time as described in Section 2. In order to execute rules according to the above criteria, a rule scheduler is needed that schedules the rules based on their priority and coupling modes. The design issues of the rule scheduler can be found in [23]. The execution of the rule scheduler and the associated data structures are explained below.

When a set of rules is triggered by an event, a thread is created for each rule that contains the rule, its priority, coupling mode, a reference to the parent thread (the thread which triggered the rule) and an operating mode. Once a rule thread is created, it can be in one of the four operating modes Ready, Wait, Exe and Finished until it finishes execution. When the thread is created, it has a 'Ready' operating mode. The thread is inserted into a data structure called the processRuleList that is shown in 

A processRuleList contains three rule queues namely, the immediateRuleQueue, deferredQueueOne and deferredQueueTwo. Each of these rule queues is a linked list of rule threads. The placement of a rule thread in the processRuleList depends on its coupling mode, parent thread and priority. Within a rule queue, the rules are inserted in the decreasing order of priority. The rules with immediate coupling mode that are triggered from the application thread are placed in the immediateRuleQueue and the rules with deferred coupling mode are placed in deferredQueueOne. It should be noted that the two deferred rule queues are interchanged when all rules in one deferred queue complete execution. When an immediate or deferred rule triggers further rules, the rules are inserted into the processRuleList as shown in the Figure 9.

The top-level rules that are triggered from the application are R1, R2 and R3. It should be noted that the parents these top-level rules are shown as 1, denoting the application thread. For rules that are triggered from the top-level rules, the name of the parent rule is shown as the parent. In the actual implementation, the reference to the parent thread is stored in each rule thread. The priorities and coupling modes are also indicated for each rule. R1 and R2 are placed in the immediateRuleQueue since they are defined with immediate coupling mode and they are stored in the decreasing order to their priority. R3 is placed in deferredQueueOne since it has a deferred coupling mode. The policy for inserting the child rules triggered from a rule is explained later in the section.

The rule scheduler is a thread running in an infinite loop that schedules i.e., starts the rule threads according to their priority and coupling modes. When the processRuleList is empty, the thread yields itself and waits. After the rule threads are created and inserted into the processRuleList in the notifyEvent method, the scheduler thread is woken up. The notifyEvent method then waits until all the rules triggered in immediate coupling mode finish execution. As mentioned in Section 2, deferred rules are executed when the event processDeffRules is raised by the application.

The scheduler checks the deferredFlag to see if the rules in immediate queue are to be scheduled or the rules in deferred queue. If the deferredFlag is false, it schedules rules in the immediateRuleQueue and if it is true, rules in deferredQueueOne are scheduled. After deciding which rules to schedule, the scheduler gets the rule threads with the highest priority from the rule queue and starts all of them simultaneously. At this time, the operating mode of the rule is changed to 'Exe'. It should be noted that rules of next

priority are to be executed only after all the higher priority rules complete execution. Thus, the scheduler waits for all the scheduled threads to complete (using a thread join) before starting the rule threads of a lower priority.

If the rule threads raise events that in turn trigger rules, the operating mode of the parent rule changes to 'Wait' and the child rules are inserted into the processRuleList. If a rule in immediate mode triggers immediate rules, the child rules are placed next to the parent rule in the immediateRuleQueue. Further, the child rules are ordered according to their decreasing priority. On the other hand, if the immediate rule triggers rules in deferred coupling mode, the rule is placed in deferredQueueOne according to its priority. The immediate rules triggered by a deferred rule are placed in the same queue as the parent and next to the parent. On the other hand, deferred rules triggered by a deferred rule are placed in the other deferred queue. As mentioned above, deferredQueueOne becomes deferredQueueTwo when all the rules in deferredQueueOne finish execution and vice-versa. After inserting the child rules in the processRuleList, the parent rule thread waits until all its child rules with immediate coupling mode, complete.

As already mentioned, the notifyEvent method waits for all the immediate rules to complete. It does this by performing a join of all the immediate rule threads. When a thread completes execution, it is removed from the processRuleList. When all the immediate rules finish execution, the scheduler thread waits, to be woken up by the notification of another event that triggers rules. When the scheduler wakes up again, it repeats the above process for the rules triggered by the new event.

## 6. CONCLUSIONS AND FUTURE WORK

This report presents an approach to incorporating active capability in a Java environment and its implementation. An event detector was implemented that detects events in Java applications and executes rules defined on them. It discusses some of the limitations of the C++ environment that influenced the design of the original version of the event detector and how we have overcome some of the limitations of the original implementation. It also discusses the alternatives available in the Java environment and the reasons behind the choice of a particular alternative.

Primitive event detection as well as composite event detection in various parameter contexts has been implemented. Unlike the earlier implementation, the current implementation uses only a single parameter list to detect events in all the four parameter contexts. This is achieved by manipulating a set of four bits (one bit for each context) associated with each event occurrence. The composite event operators that were implemented are AND, OR, SEQUENCE, NOT, APERIODIC (A), APERIODIC-STAR (A\*), PLUS, PERIODIC (P) and PERIODIC-STAR (P\*). A temporal event handler was also implemented for the relative temporal event operators PLUS, P and P\*. The temporal event handler also supports absolute temporal events as well as the specification of wild cards in absolute time expressions. The temporal event handler in C++ was platform-dependent since it used system calls to catch an interrupt from the OS timer. On the other hand, the temporal event handler in Java is platform-independent, as it does not

use system calls. The sleep method and the thread operations in Java (thread interrupt, thread wait and thread notify) are used to implement the timer.

In the earlier implementation of the event detector in C++, the event detector runs in the same thread as the application thread. In other words, all the calls to the event detector were processed synchronously in the application. This implementation separates the event detector and the application into two different threads i.e., the calls from the application to the event detector are processed in a separate thread from the application. This was primarily done to have a clean separation between the application and the event detector. We believe this will help when extending the event detector to detect events across address spaces. This also supports the parallel detection mode i.e., the detection of events can be processed either synchronously with the application or in parallel with the application. This implementation also includes a rule scheduler that schedules the rules to execute in separate threads according to their priority and coupling modes.

In the C++ version, there was only a single event graph for all the events and rules in an application whereas the current implementation allows multiple event graphs, each associated with a set of events and rules. This allows grouping of events and rules within the same application. By grouping rules, the application can disable or enable a group of rules and also use different groups of rules on the same events in different parts of the application. This will be useful for applications that require different set of rules for the same events under different circumstances.

### **6.1. Future Work**

The future work involves adding the capability of defining events and rules dynamically during application run-time without having to restart the application. Planning to use the dynamic class loading capability available in Java for achieving this. Java also has the provision for executing a block of code at class-load time. This block of code is executed only once during a single invocation of the application since a class is loaded only once into the Java virtual machine during the execution of the application. This feature is useful for creating events and rules dynamically. A graphical user interface for defining events and rules dynamically is also being planned.

Currently, the application developer has to explicitly put the raiseBeginEvent and raiseEndEvent calls inside the methods that have been defined as primitive events. Instead, a preprocessor can be used to preprocess the user classes, which puts these calls inside the methods. This is to be done before the classes are compiled along with the event detector classes. A preprocessor that parses Java classes and inserts the required calls to the event detector at appropriate places is being planned for this purpose.

The current implementation detects events only in a single application. In the future, it can be extended to detect events across multiple applications. In other words, events can be detected in a distributed system. With the use of CORBA, it can be extended to heterogeneous environments too. With the addition of some more capabilities, the event detector can also be treated as an agent that provides the service of event detection and rule execution. Other systems communicating with this agent can utilize some or all of the services provided by it.

Also, the events occurring in an application can be logged and primitive and composite event detection can be done as a post-analysis. Thus, the current implementation can be extended along multiple dimensions to enhance its functionality such that it can be utilized in the development of a wide variety of software systems that require active capability.

#### REFERENCES

- [1] S. Chakravarthy, B. Blaustein, A.P. Buchmann, M.Carey, U.Dayal, D.Goldhirsch, M.Hsu, R.Jauhari, R.Ladin, M.Livny, D.McCarthy, R.McKee, A.Rosenthal, HiPAC: A Research Project in Active, Time-Constrained Database Management, Final Report, Xerox Advanced Information Technology, Cambridge, MA, 1989.
- [2] O. D. Ray Fernandez, "Reactive Behavior Support: Themes and Variations," presented at Rules in Database Systems, Second International Workshop, Athens, Greece, September 1995.
- [3] E. Anwar, L. Maugis, and S. Chakravarthy, "A New Perspective on Rule Support for Object-Oriented Databases," in *1993 ACM SIGMOD Conference on Management of Data*, Washington, DC, 1993, pp. 99-108.
- [4] S. Chakravarthy, E. Anwar, and L. Maugis, Design and Implementation of Active Capability for an Object-Oriented Database, Technical Report, University of Florida, Gainesville, 1993.
- [5] S. Chakravarthy and D. Mishra, Snoop: An Expressive Event Specification Language for Active Databases, Technical Report, University of Florida, Gainesville, 1993.
- [6] E. Anwar, Supporting Complex Events and Rules in an OODBMS: A Seamless Approach, Master's Thesis, Database Systems R&D Center, CIS Department, University of Florida, Gainesville, 1992.
- [7] D. Mishra, SNOOP: An Event Specification Language for Active Databases, Master's Thesis, Database Systems R&D Center, CIS Department, University of Florida, Gainesville, 1991.
- [8] S. Chakravarthy and D. Mishra, An Event Specification Language (Snoop) for Active Databases and Its Detection, University of Florida, Gainesville, 1991.
- [9] V. Krishnaprasad, Event Detection for Supporting Active Capability in an OODBMS: Semantics, Architecture, and Implementation, Master's Thesis, Database Systems R&D Center, CIS Department, University of Florida, Gainesville, 1994.
- [10] N. H. Gehani, H. V. Jagadish, and O. Shmueli, COMPOSE: A System For Composite Event Specification and Detection, AT&T Bell Laboratories, Cambridge, MA, 1992.

- [11] N. Gehani and H. V. Jagadish, "Ode as an Active Database: Constraints and Triggers," in *Proceedings 17th International Conference on Very Large Data Bases*, Barcelona, 1991.
- [12] O. Diaz, N. Paton, and P. Gray, "Rule Management in Object-Oriented Databases: A Uniform Approach," in *Proceedings 17th International Conference on Very Large Data Bases*, Barcelona, 1991.
- [13] S. Gatzui and K. R. Dittrich, "Events in an Active Object-Oriented System," in *Rules in Database Systems*, N. Paton and M. Williams, Eds., Springer-Verlag, Berlin, 1993, pp. 127-142.
- [14] S. Gatzui and K. R. Dittrich, "Detecting Composite Events in Active Databases Using Petri Nets," in *Proceedings of the 4th International Workshop on Research Issues in Data Engineering: Active Database Systems*, 1994, pp. 2-9.
- [15] A. Desoto, "Using the Bean Development Kit 1.0", Tutorial, Mountain View, CA, November 1997.
- [16] <http://java.sun.com/beans/infobus/>, InfoBus, August 1999.
- [17] <http://www.vitria.com>, Vitria BusinessWare, August 1999.
- [18] <http://www.weblogic.com/docs/techoverview/em.html>, WebLogic Events Architecture, August 1999.
- [19] S. Chakravarthy, V. Krishnaprasad, E. Anwar, and S. K. Kim, "Composite Events for Active Databases: Semantics, Contexts and Detection," in *Proceedings International Conference on Very Large Data Bases*, Santiago, Chile, 1994, pp. 606-617.
- [20] S. Chakravarthy, E. Anwar, L. Maugis, and D. Mishra, "Design of Sentinel: An Object-Oriented DBMS with Event-Based Rules," *Information and Software Technology*, Vol. 36, pp. 559-568, 1994.
- [21] H. Lee, Support for Temporal Events in Sentinel: Design, Implementation, and Preprocessing, Master's Thesis, Database Systems R&D Center, University of Florida, Gainesville, 1996.
- [22] G. A. Ziemann, Detailed Design Description: Time Queue Handler, University of Florida, Gainesville, 1995.
- [23] S. Neelakantan, Scheduling Rules in an Active DBMS Using Nested Transaction Master's Thesis, University of Florida, Gainesville, 1998

***MISSION  
OF  
AFRL/INFORMATION DIRECTORATE (IF)***

*The advancement and application of Information Systems Science  
and Technology to meet Air Force unique requirements for  
Information Dominance and its transition to aerospace systems to  
meet Air Force needs.*